Solution Details: Readers

```cpp
Public Database::Read() {
    StartRead();
    Access database;
    DoneRead();
}
```

```cpp
Private Database::StartRead() {
    lock.Acquire();
    while ((AW+WW) > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.Release();
}
```

```cpp
Private Database::DoneRead() {
    lock.Acquire();
    AR--;
    if (AR ==0 && WW > 0) {
        okToWrite.signal();
    }
    lock.Release();
}
```

Solution Details: Writers

```cpp
Public Database::Write() {
    StartWrite();
    Access database;
    DoneWrite();
}
```

```cpp
Private Database::StartWrite() {
    lock.Acquire();
    while ((AW+AR) > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.Release();
}
```

```cpp
Private Database::DoneWrite() {
    lock.Acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```

Concurrency Issues

- Past lectures:
  - Problem: Safely coordinate access to shared resource
  - Solutions:
    - Use semaphores, monitors, locks, condition variables
    - Coordinate access within shared objects

- What about coordinated access across multiple objects?
  - If you are not careful, it can lead to deadlocks

- Today’s lecture:
  - What is a deadlock?
  - How can we address deadlocks?
**Deadlocks**

**Motivating Examples**

- Two producer processes share a buffer but use a different protocol for accessing the buffers
  
  ```plaintext
  Producer1() {
  P(emptyBuffer)
  P(producerMutexLock)
  }
  ```

- A postscript interpreter and a visualization program compete for memory frames
  
  ```plaintext
  PS_Interpreter() {
  request(memory_frames, 10)
  request(frame_buffer, 1)
  }
  ```

**The TENEX Case**

- If a process requests all systems buffers, operator console tries to print an error message
  
  ```plaintext
  Visualize() {
  request(frame_buffer, 1)
  request(memory_frames, 20)
  }
  ```

**Deadlock**

**Definition**

A set of processes is deadlocked when every process in the set is waiting for an event that can only be generated by some process in the set.

- Starvation vs. deadlock
  - Starvation: threads wait indefinitely (e.g., because some other thread is using a resource)
  - Deadlock: circular waiting for resources
  - Deadlock \not\equiv starvation, but not the other way

**A Graph Theoretic Model of Deadlock**

- Basic components of any resource allocation problem
  - Processes and resources
  - Model the state of a computer system as a directed graph
  
    ```plaintext
    G = (V, E)
    ```

- Where:
  - \( V \) is the set of vertices = \( \{P_1, ..., P_n\} \) (\( P_i \) is a process)
  - \( R_1, ..., R_m \) (\( R_j \) is a resource)

- \( E \) is the set of edges = \( \{\text{edges from a process to a resource}\} \)

- \( \text{edges from a process to a resource} \)
Resource Allocation Graphs

Examples

- A PostScript interpreter that is waiting for the frame buffer lock and a visualization process that is waiting for memory

\[ V = \{ \text{PS interpret}, \text{visualization} \} \subseteq \{ \text{memory frames}, \text{frame buffer lock} \} \]

A Graph Theoretic Model of Deadlock

Resource allocation graphs & deadlock

- Theorem: If a resource allocation graph does not contain a cycle then no processes are deadlocked

A cycle in a RAG is a necessary condition for deadlock

Is the existence of a cycle a sufficient condition?

A Graph Theoretic Model of Deadlock

Resource allocation graphs & deadlock

- Theorem: If there is only a single unit of all resources then a set of processes are deadlocked iff there is a cycle in the resource allocation graph

An operational definition of deadlock

- A set of processes are deadlocked iff the following conditions hold simultaneously
  1. Mutual exclusion is required for resource usage (serially usable)
  2. A process is in a "hold-and-wait" state
  3. Preemption of resource usage is not allowed
  4. Circular waiting exists (a cycle exists in the RAG)
Dealing With Deadlock
Deadlock prevention & avoidance

- Adopt some resource allocation protocol that ensures deadlock can never occur
  - Deadlock prevention/avoidance
    - Guarantee that deadlock will never occur
    - Generally breaks one of the following conditions:
      - Mutex
      - Hold-and-wait
      - No preemption
      - Circular wait
  - Deadlock detection and recovery
    - Admit the possibility of deadlock occurring and periodically check for it
    - On detecting deadlock, abort
      - Breaks the no-preemption condition